SPRAWOZDANIA
INSTYTUTU INFORMATYKI
UNIWERSYTETU WARSZAWSKIEGO

Nr 93

Piotr Findeisen
Kazimierz Jojczyk

Oprogramowanie przenośne na minikomputerze MERA-400

IInf UW REPORTS
INSTITUTE OF INFORMATICS, UNIVERSITY OF WARSAW

SIGILLUM UNIVERSITATIS VARSOVIENSIS

S P R A W O Z D A N I A

I N S T Y T U T U I N F O R M A T Y K I U W

IInf UW Reports

Nr 93

Piotr Findeisen
Kazimierz Jojczyk

Oprogramowanie przenośne na minikomputerze MERA-400
część I
Makroprocesor STAGE 2

W Y D A W N I C T W A
U N I W E R S Y T E T U W A R S Z A W S K I E G O
1 9 8 0

Adresy autorów:

P. Findeisen:
Instytut Informatyki Uniwersytetu Warszawskiego
skrytka pocztowa 1210, 00-901 Warszawa.

K. Jojczyk:
Instytut Informatyki Uniwersytetu Jagiellońskiego
ul. Kopernika 27, 31-505 Kraków

Kazimierz Jojczyk, asystent w Instytucie Informatyki U.J, przebywał w styczniu i lutym 1980 r. na praktyce zawodowej w IInf U.W.

Pracę zgłosił w dniu 5.05.1980 r.
doc. dr hab. A. Kreczmar.

Wydawnictwa Uniwersytetu Warszawskiego

Wydanie I. Nakład 300 egz. Powielono w powielarni własnej UW w październiku 1980 r. Nr GP II-441/37/80. Zam. Nr 1467/80

1. Wprowadzenie

Pod koniec lat sześćdziesiątych zostały zaprojektowane dwa konkurencyjne makroprocesory(*) ogólnego przeznaczenia: ML/1 [2] i STAGE2 [17]. Obydwa powstały między innymi dla rozwiązania pewnych problemów związanych z przenaszalnością oprogramowania.

W odróżnieniu od ML/1 pracującego w trybie ostrzegawczym (makrowołania wyróżnione są w tekście specjalnym znakiem [2]) STAGE2 pracuje w trybie swobodnym, co oznacza, że potencjalnie każda linia tekstu może być potraktowana jako makrowołanie.

W celu przedstawienia sposobu działania makroprocesora STAGE2 posłużymy się przykładem. Podane na rys. 1.1 makrodefinicje pozwalają na rozszerzenie języka Fortran o możliwości używania stałych symbolicznych. Stałe muszą być zdefiniowane przed użyciem i są identyfikowane przez nazwy poprzedzone znakiem # np. #ALFA, #MAX itp. Stałym mogą być nadawane wartości przy użyciu instrukcji podstawienia postaci:

<stała symboliczna> := <wyrażenie zbudowane ze stałych> ;

Wyrażenie budowane jest zgodnie z ogólnie przyjętymi zasadami dla konstrukcji wyrażeń fortranowskich, z tym, że wszystkie argumenty muszą być stałymi napisanymi explicite lub zdefiniowanymi uprzednio stałymi symbolicznymi. Zakłada się, że typ argumentów oraz całego wyrażenia jest INTEGER. Zamiast nawiasów () w wyrażeniu należy używać nawiasów [].

Np.:

```
#MAX := 1000;
#LIMIT := [#MAX+5]*4;
```

Nadawanie wartości stałym symbolicznym może nastąpić w dowolnym miejscu programu.

(*) U czytelnika zakłada się znajomość elementarnych pojęć związanych z makroprzetwarzaniem, takich jak makrodefinicja, makrowołanie, makrowzornik czy makrociało, np. w zakresie monografii [2].

```
;'%'0 [+-*/]
'≠':=';
!SET ≠'20 '34%
%
!SET ' ';
'F3%
%
'≠' ';
≠'20'26%
!IF '21 NE  SKIP 1%
C ******  '20 - INVALID USE'F1%
'10'21'30%
%
!IF ' NE ' SKIP ';
'F51%
%
C'END OF PROGRAM;
'F0%
%%
      ≠MAX:=100;
      ≠SIZE:=12;
      ≠DIM:=≠MAX*≠SIZE;
      ≠INPUT:=1;
      ≠OUTPUT:=2;
C
      DIMENSION DATA (≠DIM )
      LOGICAL CHECK
C
      DO 10 I=1,≠DIM ,≠SIZE
         N = I + ≠SIZE - 1
   10    READ (≠INPUT ,1000) (DATA(J),J=I,N)
 1000 FORMAT (≠SIZE A1)
      IF (CHECK (DATA)) GOTO 20
      WRITE (≠OUTPUT ,1100)
 1100 FORMAT (8H DATA OK)
      STOP
C
   20 WRITE (≠OUTPUT ,1200)
 1200 FORMAT (16H DATA NOT CORRECT)
      END
C
      LOGICAL FUNCTION CHECK (DATA)
      DIMENSION DATA (≠DIM )
      DATA N/≠DIM /
           .
           .
           .
      END
C     END OF PROGRAM
```

rys. 1.1

Przykład

Po przebiegu STAGE2 stałe symboliczne występujące w tekście fortranowskim zostaną zamienione na odpowiednie stałe numeryczne. Stałe symboliczne powinny w związku z tym wystąpić tylko tam, gdzie składnia języka Fortran dopuszcza wystąpienie stałych numerycznych. Ponadto dla ułatwienia zakładamy, że po nazwie stałej musi wystąpić odstęp.

W wyniku działania makroprocesora z danych z rys. 1.1 otrzymano tekst:

```
C
      DIMENSION DATA (1200)
      LOGICAL CHECK
C
      DO 10 I=1,1200,12
         N = I + 12- 1
   10    READ (1,1000) (DATA(J),J=I,N)
 1000 FORMAT (12A1)
      IF (CHECK (DATA)) GOTO 20
C ******  $OUTPUT - INVALID USE
      WRITE (,1100)
 1100 FORMAT (8H DATA OK)
      STOP
C
   20 WRITE (2,1200)
 1200 FORMAT (16H DATA NOT CORRECT)
      END
C
      LOGICAL FUNCTION CHECK (DATA)
      DIMENSION DATA (1200)
      DATA N/1200/
           .
           .
           .
      END
```

W pierwszej linii danych wejściowych umieszczone są zawsze kolejno znaki reprezentujące symbole:

- końca linii makrowzornika (tu %)
- parametru formalnego (')
- końca linii makrociała (%)
- parametru aktualnego (')
- cyfry zero (0)
- odstępu ( )

a następnie ujęte w nawiasy ([]) znaki operatorów arytmetycznych, odpowiednio: dodawania, odejmowania, mnożenia i dzielenia (+-*/).

W liniach (2), (5), (8), (14) i (17) występują makrowzorniki, do których STAGE2 będzie usiłował dopasować przetwarzane linie tekstu. Kolejność deklaracji makrowzorników nie ma znaczenia, zaś w przypadku niejednoznaczności wybrany zostanie wzornik najbardziej odpowiadający danemu tekstowi. Instrukcja definiująca stałą

#MAX:=100;

traktowana jako makrowołanie będzie dopasowana do makrowzornika(2). Po dopasowaniu, z tekstu makrowołania zostaną wyizolowane parametry aktualne:

- " ",
- "MAX",
- "100".

Następnie przetwarzane są kolejne linie makrociała (w tym przypadku tylko jedna linia (3)), przy czym w miejsca wystąpienia znaków parametru aktualnego ustawione są odpowiednio zmodyfikowane wartości wskazanych parametrów. Numer parametru i rodzaj modyfikacji podane są bezpośrednio za znakiem parametru aktualnego.

Na przykład '20 oznacza kopię parametru o numerze 2, zaś '34 wartość trzeciego parametru potraktowanego jako wyrażenie arytmetyczne. Tak więc, gdyby piątym parametrem był tekst "1001", a ósmym "1+19/3", to znaki '54 zostałyby zastąpione przez "1001", a '84 przez znak "7".

W wyniku przetwarzania linii (3) otrzymujemy tekst:

!SET #MAX 100

który STAGE2 znów usiłuje dopasować do któregoś makrowzornika.

W tym przypadku zostanie wybrany wzorzec (5), a nie (8) ponieważ w pierwszym z nich występuje większa liczba symoli zgodnych (!SET) niż w drugim (#).
Wartość parametru zostanie ustalona jako "#MAX" i "100".
W linii (6) umieszczone jest wołanie funkcji procesora(*) o numerze 3. Wywołanie to różni się syntaktycznie od modyfikacji parametru tym, że po znaku ' nie występuje numer parametru.

Funkcja nr 3 powoduje umieszczenie w pamięci asocjacyjnej pod wirtualnym adresem opisanym wartością pierwszego parametru łańcucha będącego wartością drugiego parametru. W naszym przypadku tekst 100 zostanie zapamiętany pod adresem #MAX. Znak końca rekordu makrociała występujący na początku linii (7) zasygnalizuje koniec przetwarzania bierzącej makrodefinicji. Sterowanie zostanie przekazane poziom wyżej, do linii (4), gdzie zostanie zakończona interpretacja ciała makrodefinicji (2).

W podobny sposób będą potraktowane linie (21) - (24). W czasie interpretacji linii (22) nastąpi obliczenie wartości wyrażenia arytmetycznego

#MAX* #SIZE.

Tutaj (czynniki nie są liczbami) jako argumenty operacji mnożenia zostaną użyte wartości zapisane pod odpowiednimi adresami w pamięci.

Ogólna zasada ewaluowania wyrażeń arytmetycznych pozwala na mieszanie argumentów bezpośrednich (liczby) i pośrednich (inne teksty), oraz używanie czterech podstawowych operacji arytmetycznych i nawiasów.

Linia (25) nie zostanie dopasowana do żadnego makrowzornika. W takim przypadku makroprocesor przekazuje ją bez zmian na wyjście.

Linia (26) zostanie dopasowana do (8). Parametrami będą odpowiednio " DIMENSION DATA (","DIM" oraz ")".

W linii 9 konstruowany zostaje tekst #DIM, a szósty typ modyfikacji (26) powoduje, że staje się on nową wartością parametru drugiego. Teraz pod adresem będącym aktualną wartością drugiego parametru powinna znajdować się wartość stałej. Następna linia ciała sprawdza, czy pod tę nazwę została już podstawiona jakaś wartość. W tym celu przetwarzana jest kolejna modyfikacja, w któ-

---

(*) Funkcją procesora nazywamy tu elementarną operację oferowaną przez jądro makroprocesora STAGE2

rej występuje porównanie zawartości pamięci o interesującym nas adresie ('21 - zawartość pamięci adresowanej drugim parametrem) ze słowem pustym. W przypadku negatywnego wyniku porównania zostanie pominięta następna linia tekstu. Mechanizm ten implementuje funkcja procesora nr 5, która porównuje teksty pierwszego i drugiego parametru, zaś wartość trzeciego parametru określa liczbę linii tekstu, przeznaczonych do ewentualnego pominięcia.

Linia (11) będzie więc przetwarzana tylko w przypadku, gdy stała symboliczna ma niezdefiniowaną wartość. Dzięki wywołaniu funkcji procesora o numerze 1 linia ta (po przetworzeniu) zostanie wypisana na wyjście bez próby dopasowania do któregoś z makrowzorników.

W linii (12) rekonstruowana zostaje instrukcja z programu źródłowego, z tym, że nazwa stałej zostanie zastąpiona przez jej wartość. O ile w tej instrukcji występuje jeszcze jedna nazwa stałej symbolicznej, zostanie ona ponownie dopasowana do makrowzornika (8). Opisany wyżej proces powtórzy się rekurencyjnie odpowiednią ilość razy.

Zakończenie pracy makroprocesora nastąpi w momencie wywołania funkcji procesora o numerze zero (F0) po napotkaniu komentarza

```
C     END OF PROGRAM.
```

Podamy tu jeszcze jeden, znacznie prostszy przykład również dotyczący języka Fortran. W wersji tego języka dla maszyn CDC można w jednej linii umieścić kilka nieetykietowanych instrukcji porozdzielanych znakiem $. Poniższa makrodefinicja realizuje zadanie standaryzacji programów wykorzystujących tę możliwość.

```
%'%'0 (+-*/)
$ '%
'10 'F1%
      '20%
%%
```

Każda przetwarzana linia wejściowa, która zawiera znak $ zostanie rozłamana tym znakiem na dwie części, z których pierwsza jest natychmiast wyprowadzana na wyjście, a druga, poprze-

dzona sześcioma spacjami (miejsce etykiety) podlega dalszemu przetwarzaniu.

Jak widać z powyższych przykładów STAGE2 dysponuje pamięcią asocjacyjną adresowaną dowolnym łancuchem znaków, w której można umieszczać również dowolne łancuchy znakowe. Dysponuje on także możliwością obliczania wartości wyrażeń arytmetycznych, zapisanych tekstowo. Ponadto w STAGE2 można korzystać z sekwencyjnych zbiorów roboczych, których liczba zależy od konkretnej implementacji, a także używać formatowanego wyjścia. Makroprocesor oferuje też generator unikalnych symboli, które można wykorzystać np. w celu tworzeniu etykiet w programie wynikowym. Specjalne funkcje procesora pozwalają na organizowanie pętli, które wykonywane są podaną ilość razy lub mogą być sterowane wskazanym łańcuchem znaków. Istnieje także możliwość wykonywania skoków warunkowych i bezwarunkowych.

Niniejsza publikacja ma na celu dość pobieżne zapoznanie czytelnika z makroprocesorem STAGE2, a także przedstawienie niektórych zagadnień związanych z jego implementacją. W ostatnim rozdziale umieszczono przegląd zastosowań STAGE2.

Czytelnika pragnącego dokładnie poznać ten makroprocesor ze strony użytkowej odsyłamy do obszernej, popartej wieloma przykładami publikacji [14] lub [8].

2. Opis typów modyfikacji parametrów

W poniższych przykładach znak D oznacza cyfrę różną od zera - numer modyfikowanego parametru. Zakładamy też, że apostrof (') jest znakiem parametru aktualnego.

Jeśli po znaku parametru aktualnego wystąpi cyfra 0, to sekwencja 'Om, gdzie m jest dowolnym znakiem, zostanie potraktowana jako żądanie generowania kolejnego unikalnego symbolu. Makroprocesor wykorzystuje w tym celu rejestr zawierający symbol generowany o początkowej wartości 1. Każde pierwsze (dynamicznie) wystąpienie znaków 'Om w danej makrodefinicji powoduje zdefiniowanie wstawki o wartości symbolu generowanego, podstawienie jej za znaki 'Om w przetwarzanym tekście i zwiększenie zawartości symbolu generowanego o 1. Każde następne wystąpienie 'Om (dla ustalonego m) w tej sa-

mej makrodefinicji powoduje już tylko podstawienie w miejsce znaków 'Om uprzednio zdefiniowanej wstawki.

'D0 Kopiowanie parametru o numerze D

Znaki 'D0 zostaną w trakcie przetwarzania zastąpione kopią parametru o numerze D.

'D1 Kopiowanie zawartości pamięci

Miejsce znaków 'D1 w przetwarzanym tekście zajmą znaki mieszczące się w pamięci asocjacyjnej pod adresem określonym przez wartość parametru D. Jeśli obiekt o takim adresie nie został jeszcze utworzony, to znaki 'D1 zostaną zastąpione łańcuchem pustym.

'D2 Kopiowanie zawartości pamięci

Jeśli obiekt o adresie będącym wartością parametru D jeszcze nie istnieje, to makroprocesor tworzy go i wypełnia wartością symbolu generowanego (wartość symbolu generowanego jest oczywiście zwiększana o 1). Dalsze postępowanie jest identyczne jak w przypadku modyfikacji nr 1.

Przykład:

Dwie podane niżej makrodefinicje rozwiązują zadanie transformacji etykiet abstrakcyjnego języka zawierającego instrukcje

```
JUMP    <label> .
LOCATE  <label> .
```

w którym etykiety (<label>) mogą być dowolnymi łańcuchami znakowymi.

```
JUMP '.
    JMP L '12%
%
LOCATE '.
L '12:%
%
```

Dane wejściowe:

```
        JUMP 10A.
        LOCATE X2.
          .
          .
        LOCATE 10A.
        JUMP X2.
          .
          .
          .
        JUMP 10A.
```

Otrzymamy wynik :

```
                JMP L1
L2:
                .
                .
L1:
                JMP L2
                .
                .
                .
                JMP L1
```

D3 Znak po parametrze

Miejsce znaków zajmie znak znajdujący się w przetwarzanym aktualnie makrowołaniu bezpośrednio za parametrem o numerze D. (Patrz również przykład po opisie D7)

D4 Wartość wyrażenia arytmetycznego

Znaki D4 zostaną zastąpione łańcuchem cyfr (obowiązuje układ dziesiętny!) będącym wartością parametru D potraktowanego jako wyrażenie arytmetyczne. W konstrukcji wyrażenia dopuszczane są cztery podstawowe operatory arytmetyczne oraz nawiasy. Argumentami mogą być liczby lub zmienne adresujące pamięć. Przyjęto, że niezdefiniowana zmienna ma wartość zero. W trakcie ewaluacji parametru o niepoprawnej budowie sygnalizowany jest błąd (niefatalny, tj. nie przerywający pracy makroprocesora)

```
******** EXPR ERROR.
```

D5 Długość łańcucha znaków

Znaki D5 są zastępowane liczbą, będącą długością łańcucha parametru o numerze D. Łańcuch pusty ma długość 0.

D6 Zmień wartość parametru

Wytworzony w trakcie przetwarzania bieżącej linii makrociała tekst stanie się nową wartością parametru o numerze D, dotychczasowa zawartość wygenerowanej linii zostaje anulowana, a makroprocesor podejmuje dalszą pracę pomijając jeden znak z przetwarzanej linii makrociała.

Parametr określony za pomocą tej modyfikacji wewnątrz pętli jest niezdefiniowany po zakończeniu bieżącego cyklu pętli.

D7 Pętla po ogranicznikach

Pojawienie się siódmego typu modyfikacji w przetwarzanej linii makrociała dzieli ją na dwie części. Pierwsza, już przetworzona, stanowi obiekt iteracji, druga zaś określa zbiór jednoznakowych ograniczników wyróżnianych w obiekcie iteracji. Obiekt ten dzielony jest na fragmenty (separowane ogranicznikami), które stają się w pętli kolejnymi wartościami parametru o numerze D. W pętli przetwarzane są linie makrociała: od bieżącej - do linii zawierającej wołanie funkcji procesora o numerze 8.

Jeśli podany zbiór ograniczników jest pusty, to iteracja przebiega po pojedyńczych znakach.

Przykład

Tłumaczenie prostych instrukcji podstawienia na assembler. Analizę tego przykładu pozostawiamy czytelnikom.

```
;'%'0 (+-*/)
LET ' = ';
ORG'46%
ORG := %
'20'37+-%
IF '41 = + SKIP 5%
IF '41 = - SKIP 2%
     SET  R1, '30'F1%
SKIP 3%
     SUB  R1, '30'F1%
SKIP 1%
     ADD  R1, '30'F1%
ORG := '33%
'F8%
     STR  R1, '10'F1%
%
IF ' = ' SKIP ';
'F50%
%
SKIP ';
'F4%
%
' := ';
'F3%
%%
```

Makrowołanie:

```
LET A = K1+ALFA-BETA.
```

wygeneruje tekst:

```
SET   R1, K1
ADD   R1, ALFA
SUB   R1, BETA
STR   R1, A
```

Jeżeli został podany pusty zbiór ograniczników, to iteracja przebiega po pojedynczych znakach.

'D8 Konwersja znaku

Znaki 'D8 zostaną w przetwarzanym tekście zastąpione liczbą będącą kodem (zależnym od implementacji) jednoznakowego parametru o numerze D. Przy modyfikacji niejednoznakowego parametru sygnalizowany jest błąd (niefatalny)

```
******** CONV ERROR
```

3. Opis funkcji procesora

F0 Zakończenie przetwarzania

Wywołanie funkcji o numerze zero powoduje zakończenie pracy makroprocesora. Jest to jedyny sposób poprawnego zakończenia obliczeń.

F1m Wypisanie linii na wyjście

Wygenerowana linia zostaje wyprowadzona bez próby dopasowania do żadnego makrowzornika na kanał o numerze m.

Jeżeli m jest opuszczone, to zostanie użyty kanał o numerze 3. Kanał o numerze zero może być użyty jako "zaślepka".

Jeśli znaki F1 pojawiły się na początku linii, to wołanie tej funkcji traktowane jest jako żądanie formatowanego wyjścia. Następna linia powinna zawierać format konstruowanego tekstu. Każda sekwencja jednakowych cyfr definiuje pole, w którym zostanie umieszczona wartość parametru (obcięta lub uzupełniona z prawej znakami spacji) o numerze wskazanym przez daną cyfrę. Np.

```
LOC '.
 F1%
11111 CONTINUE%
%
```

dla makrowołania:

```
LOC 20.
```

wygeneruje poprawną w języku Fortran instrukcję:

```
20  CONTINUE
```

n F2m Kopiowanie tekstu

Kanał o numerze n stanie się aż do momentu następnej zmiany bieżącym kanałem dla wprowadzania makrowołań.

Jeśli pierwszy parametr nie jest łańcuchem pustym, to na

kanał o numerze m zostanie skopiowany tekst z kanału wejściowego, aż do natrafienia na początku linii na znaki będące wartością pierwszego parametru.

Jeśli n lub m jest opuszczone, to w ich miejsce przyjmuje się odpowiednio wartości 2 i 3.

Po numerze kanału może wystąpić dodatkowy znak (różny od cyfry), który spowoduje wykonanie operacji przewinięcia na odpowiednim kanale przed wykonaniem transmisji.

F3 Pamiętanie łańcucha w pamięci

Wartość drugiego parametru zostaje zapisana w pamięci asocjacyjnej pod adresem będącym wartością pierwszego parametru.

F4 Skok bezwarunkowy

Pierwszy parametr, traktowany jako wyrażenie arytmetyczne, wyznacza liczbę linii, które zostaną pominięte w trakcie przetwarzania. Jeżeli liczba pozostałych linii makrociała jest mniejsza od wartości wyrażenia, to opuszczone zostaną (w odpowiedniej ilości) linie makrowołań lub makrodefinicji wywołujących daną.

F5k Porównanie łańcuchów znakowych

Porównywane są łańcuchy znakowe będące wartościami dwóch pierwszych parametrów. Liczbę pomijanych linii wyznacza parametr trzeci traktowany jako wyrażenie arytmetyczne. O typie porównania decyduje znak k występujący bezpośrednio za znakami F5. Skok jest efektywny, gdy:

k = 0 i łańcuchy są równe
k ≠ 0 i łańcuchy są różne.

F6k Porównanie wartości arytmetycznych

Porównywane są wartości dwóch pierwszych parametrów traktowanych jako wyrażenia arytmetyczne. Parametr trzeci wyznacza liczbę przeskakiwanych linii. Skok jest efektywny, gdy:

k = - i parametr 1 < parametr 2
k = 0 i parametr 1 = parametr 2
k = 1 i parametr 1 ≠ parametr 2
k = + i parametr 1 > parametr 2.

'F7 Pętla z licznikiem

Linie makrociała znajdujące się między bieżącą linią, a końcem pętli sygnalizowanym przez 'F8 przetwarzane są w pętli tyle razy , ile wynosi wartość wyrażenia arytmetycznego znajdującego się przed 'F7.

'F8 Ogranicznik pętli

Linia zawierająca na początku znaki 'F8 stanowi statyczne ograniczenie dla sekwencji lini przetwarzanych w pętli organizowanej przez 'D7 lub 'F7.

'F9 Zakończenie przetwarzania bieżącej makrodefinicji

Po napotkaniu wołania funkcji nr 9 makroprocesor kończy przetwarzanie bieżącej makrodefinicji.

'FE Drukuj ślad

Po napotkaniu w rekordzie znaków 'FE zostaje zasygnalizowany błąd

******* CONV ERROR

i wprowadzony zostaje ślad przetwarzania aktualnego makrowołania w postaci listy wywołań, które doprowadziły do analizowania bieżącego rekordu makrociała.

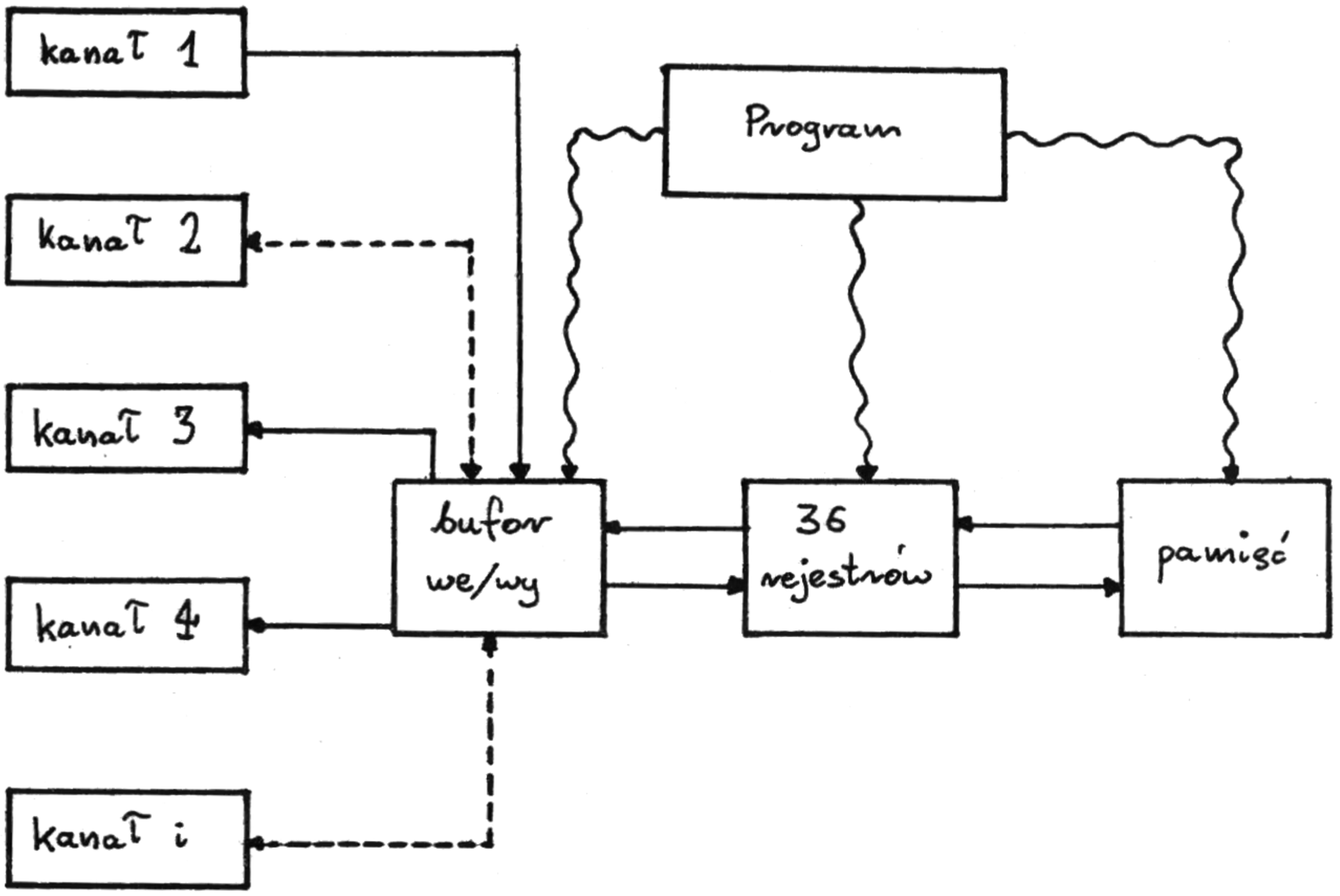
Szczegółowy opis typów modyfikacji i funkcji procesora można znaleźć w [6], [8], [14] i w oryginalnej pracy Waite 'a [17].

4. Maszyna abstrakcyjna FLUB

Makroprocesor STAGE2 został napisany w języku abstrakcyjnym maszyny FLUB (First Language Under Bootstrap) [17]. Maszyna ta składa się z 36 rejestrów, z których każdy jest podzielony na trzy części o nazwach FLG, VAL i PTR, pamięci zawierającej słowa o takiej samej jak rejestry konstrukcji, bufora we/wy, oraz pewnej liczby abstrakcyjnych kanałów we/wy. Każde z pól rejestrów dostępne jest niezależnie od pozostałych. Kanały o numerach 1, 3, 4 występują zawsze i mają ustalone przeznaczenie:

1. kanał wejściowy
3. kanał wyjściowy
4. kanał komunikatów operatorskich.

Pozostałe kanały mogą występować w zależności od potrzeb użytkownika.



rys. 4.1
Konfiguracja maszyny abstrakcyjnej FLUB

W części VAL rejestrów przechowywane będą znaki lub niewielkie liczby całkowite określające długość łańcucha znakowego. Ponadto wartością pola VAL może być -1.

Części PTR używane są do adresowania pamięci maszyny FLUB, a także do obliczania wartości wyrażeń arytmetycznych.

Część FLG przeznaczona jest wyłącznie do markowania i może przybierać tylko cztery wartości: 0, 1, 2, 3.

Niektóre rejestry przeznaczone są do przechowywania stałych i żaden z programów napisanych w języku FLUB nie powinien ich zmieniać. Wykorzystanie rejestrów pokazuje tabela na rys. 4.2.

	FLG	VAL	PTR
0	0	0	0
1	1	1	1
2	2	2	2
3	3	3	3
4		4	
5		5	10
6		6	
7		7	X
8		8	X
9		9	X

rys. 4.2
Rozmieszczenie stałych
w rejestrach FLUB

PTR 7, PTR 8 i PTR 9 zawierają stałe zależne od implementacji i opisują pewne parametry pamięci maszyny FLUB. PTR 7 charakteryzuje rozmieszczenie pól FLG, VAL i PTR w rzeczywistej pamięci maszyny i równe jest liczbie słów maszynowych przypadających na jednostkę adresacji FLUB. Zakłada się, że adresy FLUB zmieniają się w przedziale od PTR 8 do PTR 9 z krokiem PTR 7. Tak więc różnica PTR 9 - PTR 8 określa rozmiar pamięci komputera przeznaczonej na pamięć maszyny FLUB.

Ze względu na znaczną liczbę rejestrów maszyny FLUB rejestry te nie mogą być utożsamione z fizycznymi rejestrami minikomputera. Ponieważ program odwołuje się do pól rejestrów dość często, każde z nich w opisywanej implementacji umieszczone zostało w oddzielnym słowie pamięci. Można więc utożsamiać je ze zmiennymi prostymi i nazywać VAL A, PTR B czy FLG 0.

W języku FLUB występuje 28 instrukcji. Są wśród nich operacje arytmetyczne, sterujące, przesłania między rejestrami i pamięcią, a także przesłania z/do bufora i instrukcje we/wy. Wszystkie instrukcje, z wyjątkiem tych ostatnich zostały opisane w sposób całkowicie niezależny od maszyny. Instrukcje we/wy, które powodują wykonanie transmisji między buforem a rzeczywistym oto-

czeniem implementowanej maszyny abstrakcyjnej muszą być dostosowane do konkretnego systemu cyfrowego. Jeżeli jednak na wyposażeniu tego ostatniego znajduje się translator języka Fortran, to można skorzystać z "uniwersalnego" modułu we/wy napisanego przez Waite'a [18].

5. Implementacja makroprocesora STAGE2 na minikomputerze MERA 400

Zgodnie z intencjami swoich twórców STAGE2 powinien być implementowany w dwóch krokach. W pierwszym wykorzystuje się prosty makroprocesor, nazwany przez autorów SIMCMP (Simple Common Macro Processor), który można bardzo łatwo zaprogramować w Fortranie (około 70 instrukcji) [18], a którego działanie jest podobne do działania STAGE2. Z bogatych możliwości STAGE2 pozostaje w nim jedynie kopiowanie parametrów (muszą być one dokładnie jednoznakowymi łancuchami) oraz możliwość generowania unikalnych symboli. Te możliwości okazują się wystarczające do przetłumaczenia instrukcji FLUB na wybrany język programowania.

Pełen zestaw instrukcji traktowanych jako makrowzorniki wraz z ciałami makrodefinicji tłumaczącymi je na Fortran znajduje się w dodatku A1. Makrodefinicje te można traktować jako kompletny opis działania maszyny abstrakcyjnej.

Pierwszy etap implementacji polega więc na przetłumaczeniu w dość prosty sposób tekstu makroprocesora STAGE2 (napisanego w języku FLUB) na jakikolwiek język dostępny na docelowym komputerze. Po kompilacji otrzymanego tekstu i dołączeniu do niego modułu we/wy uzyskujemy pierwotną wersję makroprocesora. Często uzyskany w ten sposób program jest nieefektywny, tak pod względem czasu działania, jak i zużycia pamięci. Dlatego w drugim etapie implementacji przygotowuje się, z wykorzystaniem pełnych możliwości STAGE2, nowe makrodefinicje tłumaczące FLUB. W obydwu etapach dużą pomocą mogą się okazać testy maszyny FLUB dostarczone przez autorów [18].

Korzystając wyłącznie z możliwości makroprocesora SIMCMP nie można zoptymalizować tłumaczonego programu. Możliwości optymalizacji jest zaś wiele.

Rozważmy np. instrukcję:

VAL K = K + 1.

Powoduje ona zwiększenie zawartości pola VAL rejestru K o zawar-

tość pola VAL rejestru 1. Zauważmy, że w polu VAL pierwszego rejestru znajduje się stała 1. Można więc w miejsce instrukcji dodającej zawartość słowa pamięci umieścić instrukcję dodającą bezpośrednio stałą. Zazwyczaj taka instrukcja działa szybciej, a w niektórych komputerach (MERA 400) zajmuje także mniej miejsca (jedno słowo, podczas gdy instrukcja dodająca argument z pamięci zajmuje dwa słowa).

Oczywiście ta sama uwaga dotyczy pozostałych stałych. Jeszcze dalej posuniętą optymalizację można zastosować przy instrukcjach przesyłających zawartości rejestrów FLUB, które są zapisywane jako dodawanie rejestru zerowego, np.:

```
PTR X = Y + 0.
```

W tym wypadku, ponieważ wartością PTR 0 jest stała zero, operację dodawania można zignorować.

W tekście makroprocesora STAGE2 często występują operacje porównania z rejestrami stałymi. I tu także udaje się wykorzystać instrukcje jednosłowowe.

Zauważmy, że wartość rejestru PTR 7, nieustalona wprawdzie w maszynie FLUB, może być uważana za stałą w każdej konkretnej implementacji. Pozwala to wykorzystać przy implementacji te same mechanizmy, które są używane przy rejestrach stałych. Ponieważ w opisanej implementacji przyjęto PTR 7 = 1, pozwoliło to uniknąć wykonywania operacji arytmetycznych przy instrukcjach mnożenia i dzielenia przez PTR 7.

Rozważmy teraz ciąg instrukcji:

```
VAL A = A + B.
VAL C = A + D.
```

W wyniku tłumaczenia pierwszej instrukcji otrzymamy sekwencję rozkazów maszynowych, powodującą umieszczenie wartości rejestru VAL A w którymś z rejestrów minikomputera, dodanie doń zawartości VAL B i zapamiętanie wyniku pod adresem VAL A. Następna instrukcja spowoduje wykonanie analogicznych czynności. Jednak ponowne ładowanie zmiennej VAL A do rejestru nie jest niezbędne, ponieważ aktualna wartość zmiennej znajduje się już w którymś z nich. Wystarczy więc dodać do niego wartość VAL D i zapamiętać

uzyskany wynik w VAL C. W wyniku takich operacji w którymś rejestrze znajdzie się aktualna wartość zmiennej VAL C.

Zauważmy, że w innych rejestrach również mogą znajdować się interesujące nas wielkości. Gdyby wśród nich znajdowała się wartość VAL B lub VAL D, to zaistniałaby możliwość użycia instrukcji dodającej zawartość rejestru do rejestru (instrukcja krótka). Opłaca się więc w trakcie tłumaczenia programu napisanego w języku FLUB na język wewnętrzny maszyny wielorejestrowej pamiętać aktualne zawartości rejestrów.

Przypuśćmy teraz, że należy obliczyć sumę trzech składników: PTR A + PTR B + PTR C i wynik umieścić w PTR X. Nie można tego zrobić jedną instrukcją FLUB. Sumę tę obliczy sekwencja:

PTR X = A + B.
PTR X = X + C.

Nawet przy zastosowaniu wyżej opisanej metody optymalizacji wykona się w tym przypadku niepotrzebna operacja pamiętania sumy A + B w PTR X. Można więc opóźniać odsyłanie zawartości rejestrów maszynowych do odpowiadających im zmiennych, a jedynie pamiętać o konieczności dokonania przesłania przed następnym (niszczącym) użyciem rejestru.

Przy takim rozwiązaniu pojawia się zagadnienie przydziału i wywłaszczania rejestrów. Aby wykonać jakąkolwiek operację korzystającą z argumentów należy przynajmniej jeden z nich umieścić w rejestrze. Ponieważ liczba rejestrów jest skończona, może zajść sytuacja, w której trzeba będzie opróżnić jeden z rejestrów z przechowywanej tam wartości, odsyłając ją do pamięci na wskazane miejsce. Przyjęty przez nas algorytm wyboru rejestru wygląda następująco:

- jeżeli któryś z rejestrów jest wolny (ma nieokreśloną zawartość), to zostanie wybrany
- jeśli żaden z rejestrów nie jest wolny, to szukamy rejestru, którego zawartość nie musi być odsyłana do pamięci (po załadowaniu rejestru jego wartość nie uległa zmianie)
- w przeciwnym wypadku wybór rejestru odbywa się zgodnie z regulaminem cyklicznym, a zwartość rejestru jest odsyłana na odpowiednie miejsce w pamięci.

Otrzymany w ten sposób algorytm optymalizacji ma wyraźne cechy statyczne. W przypadku napotkania etykiety zawartości rejestrów muszą zostać odesłane do pamięci (tam, gdzie jest to konieczne) zaś wszystkie rejestry muszą przyjąć stan "wolny", ponieważ w jednoprzebiegowym makroprocesorze nie można określić żadnych innych warunków, jakie mają być spełnione w takim punkcie programu. Dzieje się tak dlatego, ponieważ odwołania do etykiet mogą występować w dowolnych miejscach programu, których makroprocesor jeszcze nie analizował, a po napotkaniu odwołania do etykiety uprzednio zdefiniowanej nie można już zmodyfikować wygenerowanego tekstu.

Podobne algorytmy statycznej optymalizacji proponują w swych pracach Aho i Gries [1], [5].

W związku z przyjętą zasadą statycznej optymalizacji przed wykonaniem operacji skoku wartości zmiennych muszą zostać uaktualnione. Jeżeli jednak mamy wykonać skok warunkowy, pamiętanie odpowiednich rejestrów odbywać się może w zależności od tego warunku. Powoduje to jednak na ogół zwiększenie objętości programu wynikowego, bowiem instrukcje pamiętające muszą być umieszczane w tekście kilkakrotnie. Alternatywnym rozwiązaniem może być bezwzględne pamiętanie rejestrów przed sprawdzeniem warunku. Otrzymamy wówczas krótszy, ale za to nieco mniej efektywny program.

Warunkowe pamiętanie rejestrów realizowane jest przez testowanie warunku przeciwnego i przeskok instrukcji pamiętających wraz ze skokiem (już bezwarunkowym) do wskazanej etykiety.

Przykładowo fragment tekstu, napisany w języku FLUB:

```
VAL A = B + C.
VAL X = X + 1.
TO 20 IF VAL Y = X.
TO 30 IF VAL Y NE Z.
VAL A = A + Y.
```

może zostać przetłumaczony między innymi na dwa sposoby. Z lewej strony podany został tekst otrzymany przez zwykłe tłumaczenie makroprocesorem SIMCMP, zaś z prawej, po zastosowaniu optymalizacji (użytej także w ostatecznej wersji makrodefinicji).

```
SET   R1, VALB          SET   R4, VALB
ADD   R1, VALC          ADD   R4, VALC
STR   R1, VALA

SET   R1, VALX          SET   R6, VALX
ADD   R1, VAL1          ADS   R6, 1
STR   R1, VALX

SET   R1, VALY          SET   R5, VALY
CMPA  R1, VALX          CMPA  R5, R6
BEQ   L20               STR   R4, VALA
                        STR   R6, VALX
                        BEQ   L20

SET   R1, VALY          CMPA  R5, VALZ
CMPA  R1, VALZ          BNE   L30
BNE   L30

SET   R1, VALA          ADD   R4, R5
ADD   R1, VALY
STR   R1, VALA
```

Powyższe fragmenty programów zajmują w pamięci odpowiednio 30 i 21 słów. Podobną proporcję można uzyskać przy tłumaczeniu tekstu całego makroprocesora.

Jednak równie ważną jak optymalizacja przydziału rejestrów rolę dla efektywności uzyskanego programu stanowi sposób implementacji pamięci maszyny FLUB. Niech <reg1> i <reg2> będą nazwami rejestrów maszyny FLUB. Na pamięci określone są dwie operacje:

GET <reg1> = <reg2>.

powodująca odczyt wartości zapisanych w pamięci adresowej przez PTR <reg2> i podstawienie jej na pola FLG, VAL i PTR rejestru <reg1>, oraz

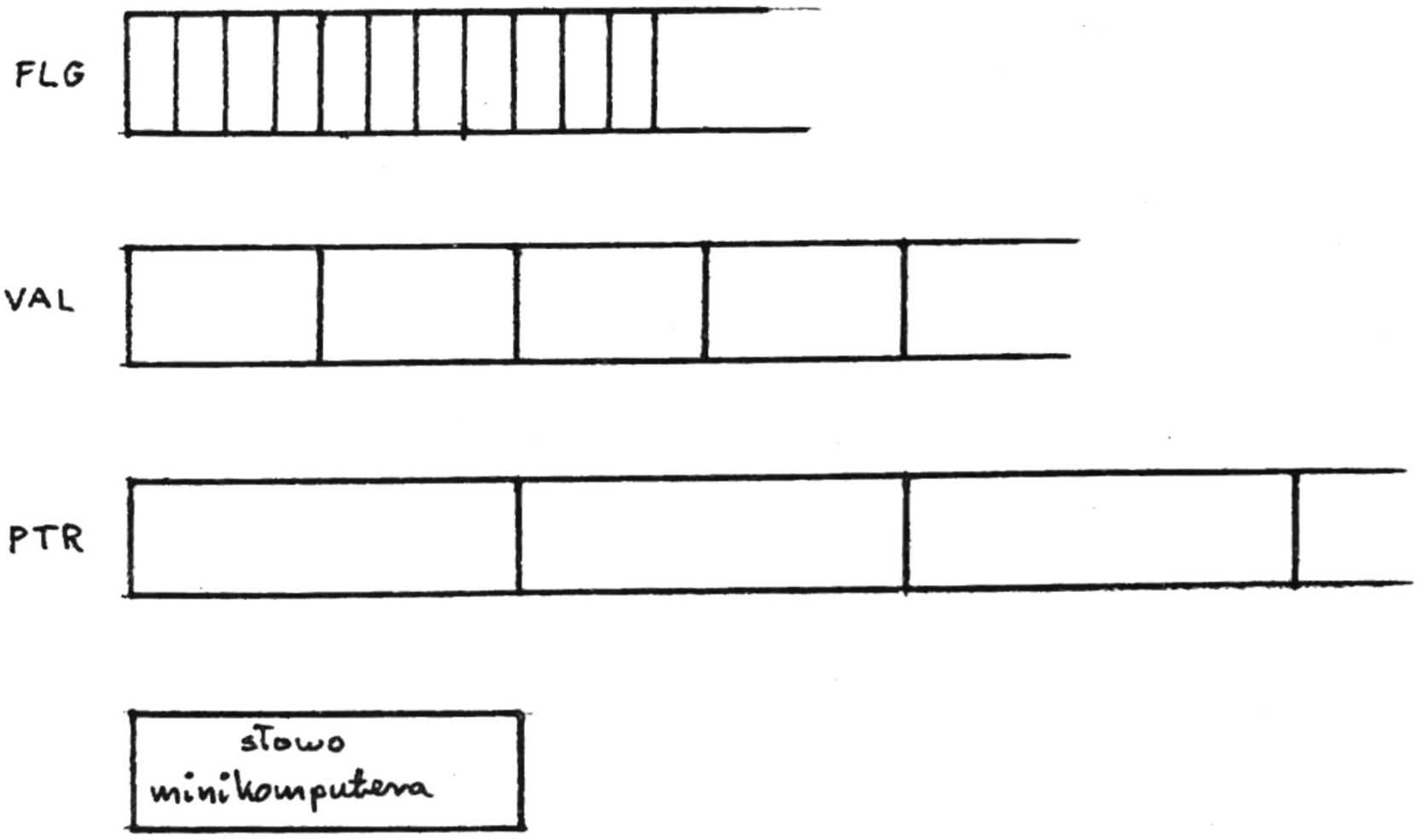
SET <reg1> = <reg2>.

zapisująca wszystkie pola rejestru <reg2> w pamięci, pod adresem PTR <reg1>.

Widać więc, że operacje te odwołują się do wszystkich trzech pól słowa maszyny FLUB za pomocą jednego, wspólnego adresu. Fizycznie rozmieszczenie pól słowa w pamięci rzeczywistej może być zorganizowane praktycznie dowolnie. Koncepcja ta umożliwia implementację pamięci w sposób maksymalnie odpowiadający potrzebom użytkownika, pozwala także na łatwe stronicowanie i wykorzystanie pamięci masowej.

Dla potrzeb implementacji makroprocesora STAGE2 na minikomputerze MERA 400 przyjęliśmy, że pole FLG zajmuje 2 bity, pole VAL - 8 bitów, a pole PTR 16 bitów. Wielkości te wynikają z fizycznych parametrów minikomputera oraz funkcji, jakie pola te spełniają w maszynie FLUB.

Aby maksymalnie wykorzystać pamięć operacyjną, można umieszczać w jednym słowie maszynowym 8 pól FLG lub 2 pola VAL, łącząc słowa przeznaczone na jeden rodzaj pól w spójne bloki.



rys. 5.1
Rozmieszczenie pól FLG, VAL i PTR
w pamięci

Przy tym rozwiązaniu operacje GET i STO okazały się dość czasochłonne. Zwłaszcza kosztownym okazało się rozpakowanie pól FLG i konieczność rozszerzenia znaku pola VAL.

Konkurencyjnym rozwiązaniem jest pamiętanie słowa FLUB w trzech kolejnych słowach pamięci. To rozwiązanie okazało się oczywiście najszybszym, ale też kosztownym pod względem pamięciowym.

Pośrednim rozwiązaniem jest przechowanie pól VAL i PTR tak, jak w sposobie pierwszym, a pól FLG - po jednym w bajcie. Wprawdzie nadal pozostaje konieczność rozszerzenia znaku pola VAL, ale czas dostępu do pól FLG, dzięki zastosowaniu rozkazów bajtowych, uległ znacznej poprawie.

Ta metoda organizacji pamięci została potraktowana przez autorów implementacji jako podstawowa (S). Zrealizowano jednak obie pozostałe wersje (L oraz F) przeprowadzając jednocześnie ich porównania.

W poniższej tabeli przedstawiona jest również pierwsza, bootstrappowa wersja makroprocesora (B), uzyskana za pomocą programu SIMCMP. W zestawieniu ujęto kolejno parametry:

1. długość otrzymanego kodu makroprocesora (bez modułu we/wy) w słowach
2. maksymalną liczbę słów pamięci FLUB, jaką można uzyskać dla danej wersji w standardowej konfiguracji minikomputera MERA 400
3. czas działania wersji na pewnym prostym przykładzie
4. czas działania makroprocesora podczas przetwarzania makrodefinicji tworzących kod dla wersji F.

Moduł we/wy został napisany dla wszystkich wersji w assemblerze i zajmuje wraz z buforami 2200 słów.

Wnioski wynikające z analizy pozycji tabeli nie są zaskakujące. Potwierdzają one regułę "trade-off" między kosztem pamięciowym, a czasowym programów.

Parametr	Wersja			
	B	L	S	F
1	5150	4500	5200	4000
2	8000	8400	6400	4750
3	65 s	60 s	38 s	31 s
4	810s	735s	510s	340s

rys. 5.2
Porównanie otrzymanych wersji makroprocesora

6. STAGE2 jako język o zredukowanym jądrze

Z pewnego punktu widzenia ulegające przetwarzaniu dane wejściowe do makroprocesora można uważać za program. Składnia języka, w którym jest napisany ów program nie jest ustalona a priori - określona jest dopiero poprzez makrodefinicje. Oczywiście opisują one również semantykę tego programu. Inny zestaw makrodefinicji mógłby np. przy niezmienionej składni opisać zupełnie inną semantykę.

Języki posiadające takie cechy zostały nazwane przez Kostera [3] językami o otwartym jądrze (open-ended). Cechują się one względnie łatwą implementacją (poprzez bootstrapping czy makroprzetwarzanie) i szeroką możliwością rozszerzania składni. Ich semantyka nie jest opisywana w żadnych terminach absolutnych, a bazuje na pewnych prymitywach zapożyczonych z innego, zdefiniowanego środowiska.

Klasycznym przykładem takiego języka jest CDL2 [9] zaprojektowany w Katholieke Universiteit, Nijmegen. W języku tym określone zostały tylko elementarne reguły dotyczące składni i semantyki, zaś treść semantyczna wszystkich konstrukcji zapożyczona jest poprzez makro-mechanizm z innego środowiska. Umożliwia to tworzenie hierarchicznych warstw (layers), z których tylko najniższe są zależne od maszyny [15].

W STAGE2 sytuacja przedstawia się podobnie. Rozważmy np. modyfikację o makrowzorniku:

GET <arg>.

która ma za zadanie umieścić podany argument <arg> w którymś z rejestrów maszyny. Argumentem może być nazwa pola konkretnego rejestru FLUB (VAL A, PTR 6 itp.).

Zgodnie z przyjętymi zasadami optymalizacji powinniśmy sprawdzić, czy żądany argument nie znajduje się już w jednym z rejestrów. Jeżeli tak jest (w pamięci asocjacyjnej pod nazwą rejestru zapisana jest wówczas nazwa argumentu), to nie należy wykonywać żadnych dodatkowych czynności.

```
GET '.
234567 '27%                    pętla po rejestrach
REG := '20%                    aktualnie rozważany rejestr
IF S( '20) = '10 SKIP 4%       czy zawiera argument?
 F8%                           nie, sprawdź następny
GETREG%                        znajdź wolny rejestr
REG 36%
   SET R 31, '10 F1%           ładuj argument
%                              nr rejestru zapamiętany w REG
```

W przypadku nieznalezienia argumentu korzystamy z makrodefinicji GETREG, dostarczonej przez niższą warstwę, GETREG powoduje zwolnienie wybranego przez siebie rejestru i podstawienie jego numeru pod adres REG.

Korzystamy tu również z dwóch innych mechanizmów opisanych przez makrodefinicje pochodzące z najniższej warstwy:

```
IF S( ') = 'SKIP'.             jeśli zawartość pamięci o
 '11 '16%                      adr. wskazanym 1 argumentem
 F50%                          równa jest 2 arg., to przeskocz
%                              liczbę linii wsk. 3 arg.

' := '.                        umieść wartość 2 argumentu
'F3%                           pod adresem wskazanym przez 1
%                              argument.
```

Makrodefinicja GET wykorzystana jest w procesie generacji optymalizowanego kodu do umieszczania w rejestrze pierwszego argumentu operacji maszynowej. Podobnie realizowane są inne operacje pomocnicze, takie jak:

STORE <machine reg> <arg>.

powodująca zaznaczenie, że zawartość rejestru <machine reg> należy odsyłać do pamięci pod adres wyznaczony przez <arg>. Zgodnie z ogólną koncepcją fizyczny zapis do pamięci odbędzie się w nieokreślonym czasie. Jednak z punktu widzenia wewnętrznej logiki warstwy wywołującej STORE zawartość rejestru można uważać za zapamiętaną.

Poprzez definiowanie odpowiednich modyfikacji zostało rozszerzone jądro makroprocesora STAGE2. Obok modyfikacji parametru i wołania funkcji procesora można teraz używać także wygodniejszych nowych operacji, które pełnią podobną pod względem funkcjonalnym rolę.

Nowe makrodefinicje konstruowane były tak, aby realizowały określone funkcje, w pewnym stopniu niezależne od maszyny. Makrodefinicja GET np. wykorzystana wszędzie tam, gdzie trzeba do rejestru maszyny załadować wskazane pole rejestru FLUBu ma funkcję określoną niezależnie od liczby rejestrów maszyny. Przy zmianie dostępnej ilości rejestrów maszynowych wystarczy dokonać stosownych zmian w GET (i również w niższych warstwach) bez zmiany kontekstu, w którym wołanie GET występuje. Podobnie makrodefinicja STORE może mieć zmienioną treść i powodować bezpośrednie wygenerowanie instrukcji pamiętającej.

7. Zastosowanie

Jak to wskazuje nazwa STAGE2 - makroprocesor ten wykorzystany był przez jego autorów jako drugi stopień bootstrappingu w procesie otrzymywania specjalizowanych produktów programowych. Analizując jego efektywność dochodzimy do wniosku, że rzeczywiście nie powinien być on wykorzystywany jako program użytkowy. Cechy, jakimi obdarzyli go autorzy czynią zaś z niego doskonałe narzędzie do implementacji maszyn ab-

strakcyjnych. (Brown w [2] wyspecyfikował klasę języków dobrze implementowanych przez makroprocesory nazywając ją DLIMP - Descriptive Languages Implemented by Macro Processor).

W pracy [18] opisano implementację kompilatora/interpretera HELP - języka podobnego do Lispu, o ograniczonych w stosunku do niego możliwościach, ale o bardziej naturalnej notacji. W języku tym ciąg wyrażeń, po uprzednim sprawdzeniu syntaktycznej poprawności, zapisywany jest przez kompilator w postaci pewnej struktury listowej. Zadaniem interpretera jest ewaluowanie kolejnych struktur i wypisywanie wyników. Tekst kompilatora/interpretera został napisany w języku maszyny abstrakcyjnej Basic Wisp [11], pozwalającym na łatwy zapis struktur listowych oraz operujących na nich algorytmów. Narzędziem implementującym tę maszynę (3 stopień bootstrappingu) jest BASCMP (Basic Wisp Macro Processor) - zmodyfikowany nieco makroprocesor STAGE2. Korzystając z dostarczonych przez autorów tekstów w trzecim kroku uzyskujemy prawie automatycznie kompilator/interpreter języka HELP.

W Ecole Royale Militaire w Brukseli prowadzone były prace nad konstrukcją analizatora syntaktycznego dla kompilatorów obszernego podzbioru języka Algol 68 [4]. W ramach eksperymentu zakodowano algorytm analizatora w prostym języku POPAC używającym notacji zgodnej z Fortranem. Programy były uruchamiane i testowane z wykorzystaniem kompilatora Fortranu. Następnie za pomocą makroprocesora STAGE2 przetłumaczono język POPAC na assemblery IBM/360 i CDC 6400. Dzięki zastosowaniu makrodefinicji w wysokim stopniu optymalizujących kod uzyskane produkty stały się mniejsze i szybsze od swoich pierwotnych odpowiedników. Wiązało się to jednak z jednorazowym, ale dość poważnym kosztem - makroprocesor pracował przez ponad 8 godzin generując kod w assemblerze dla komputera IBM 360/30.

W wyniku tego eksperymentu otrzymano, przy niewiele zwiększonym nakładzie pracy, przenaszalność analizatora syntaktycznego.

Mobilny Preprocesor Fortranu Strukturalnego (MPFS), zrealizowany w Instytucie Informatyki Uniwersytetu Jagiellońskiego pozwala na rozszerzenie języka Fortran o konstrukcje strukturalne typu IF-THEN-ELSE, REPEAT-UNTIL, WHILE-DO czy CASE,

e tym samym na czytelny zapis warunkowo wykonywanych bloków instrukcji [7]. W celu uzyskania produktu mobilnego tekst procesora został zakodowany w rozszerzonym nieco języku FLUB [18]. Preprocesor na wejściu akceptuje program napisany w rozszerzonym Fortranie i tłumaczy go na język akceptowany przez kompilator Fortranu. Rozwinięcie konstrukcji strukturalnych nie powoduje pojawienia się elementów niestandardowych.

Obliczenia preprocesora różnią się od przebiegu STAGE2 tym, że makrodefinicje nie są wczytywane przy kolejnym przebiegu, e są wbudowane w kod programu. Mimo tego istnieje możliwość zmiany tych makrodefinicji - uzyskuje się wówczas kolejne rozszerzenie akceptowanego języka.

MPPS został zaimplementowany na takich maszynach jak CYBER 72 i Odra 1306, stając się w obu wypadkach produktem o zadawalającej szybkości. STAGE2 został w tym przypadku użyty zgodnie z intencjami swoich autorów - jako pośredni stopień bootstrappingu i jego efektywność nie miała wpływu na parametry uzyskanych programów.

Przykład bezpośredniego zastosowania makroprocesora STAGE2 jako programu użytkowego można znaleźć w [12] i [13]. W pierwszej z wymienionych prac STAGE2 tłumaczy na język Fortran program napisany w Fortranie rozszerzonym o możliwość rekursji. W drugiej jest metatranslatorem sterowanym gramatykami atrybutywnymi. W tym przypadku cztery kolejne fazy: uzyskanie parsera, opisu drzewa wywodu i dwukrokowej analizy semantycznej polegają na wygenerowaniu kolejnego zestawu makrodefinicji i makrowołań dla STAGE2. W obu przypadkach postawiono na mobilność i szybkie rozwiązanie zadania, godząc się z bardzo niską efektywnością uzyskanych produktów.

W pracach Instytutu Informatyki Uniwersytetu Warszawskiego znalazło się między innymi opracowanie assemblera dla minikomputera MERA 400. Aby zapewnić dobrą dokumentację, czytelność i modyfikowalność assemblera, jego translator pisany jest w języku przez siebie akceptowanym. Makroprocesor STAGE2 tłumaczy program napisany w assemblerze na język MACROASSEMBLER opracowany przez firmę MERA. Ponieważ do assemblera wprowadzono wpływające na semantykę instrukcji typy: adresowy i numerycz-

ny, makroprocesor musi zorganizować i obsłużyć tablicę symboli. Dokonywana jest również kontrola liczby i typów argumentów instrukcji. W tym przypadku użycie STAGE2 odbywa się zgodnie z jego przeznaczeniem - krok pośredni przy implementacji assemblera.

Projekt będący między innymi próbą możliwości STAGE2, tłumaczenie z języka o większym bogactwie struktur danych - Assembler/360, na język o możliwościach mniejszych - język PLAN 4, realizowany był w ramach pracy magisterskiej w Instytucie Informatyki Uniwersytetu Jagiellońskiego [10]. Tłumaczenie - z konieczności więcej niż jednoprzebiegowe, zostało po kilku miesiącach pracy zakończone sukcesem. Otrzymany produkt finalny działający na maszynach serii Odra 1300 pracował jednak bardzo wolno, a to z powodu użycia STAGE2 - makroprocesora o przeznaczeniu ogólnym - jako narzędzia wyspecjalizowanego, o bardzo szczególnym profilu działania, jak również ze względu na małą moc obliczeniową komputera.

W planach Instytutów Informatyki Uniwersytetów Warszawskiego i Jagiellońskiego znajdują się dalsze prace związane z przenoszeniem oprogramowania mobilnego na minikomputer MERA 400.

W fazie realizacji znajduje się obecnie kompilator języka Pascal, napisany w PCODE przez U. Ammana. PCODE(język maszyny abstrakcyjnej) jest również językiem wyjściowym kompilatora. Aby więc uzyskać możliwość kompilacji i wykonywania programów wystarczy napisać np. interpreter języka PCODE.

Rozpoczęto również implementację języka CDL2 [9]. Oparta jest ona o wersję 8.1 kompilatora tego języka (H. Stahl), w której został wyraźnie wyodrębniony moduł generatora kodu. Dzięki wymianie tego modułu można, przez prosty bootstrapping, uzyskać kompilator działający na minikomuterze MERA 400.

Rozważane są także możliwości realizacji kompilatora/interpretera języka HELP [18], Prolog oraz BCPL.

A1. Składnia instrukcji maszyny abstrakcyjnej FLUB

reg - Nazwa rejestru. Nazwy są jednoznakowe. Mogą to być cyfry lub litery.

lab - Etykieta. Etykietę tworzą dwie cyfry dziesiętne.

c - Dowolny znak.

a. Operacje transmisji danych

GET reg = reg.
STO reg = reg.
FLG reg = reg.
VAL reg = PTR reg.
PTR reg = VAL reg.

b. Operacje arytmetyczne

VAL reg = reg + reg.
VAL reg = reg - reg.
PTR reg = reg + reg.
PTR reg = reg - reg.
PTR reg = reg * reg.
PTR reg = reg / reg.

c. Operacje sterujące

TO lab.
STOP.
TO lab IF FLG reg = reg.
TO lab IF FLG reg NE reg.
TO lab IF VAL reg = reg.
TO lab IF VAL reg NE reg.
TO lab BY reg.
RETURN BY reg.
TO lab IF PTR reg = reg.
TO lab IF PTR reg NE reg.
TO lab IF PTR reg GE reg.

d. Operacje we/wy

VAL reg = CHAR.
CHAR = VAL reg.
MESSAGE cccc TO reg.
READ NEXT reg.
WRITE NEXT reg.
REWIND reg.

e. Pseudooperacje

LOC lab.
END PROGRAM.

Poniżej przedstawione są makrodefinicje i początkowy fragment procedury stanowiącej po przetworzeniu przez SIMCMP zapisany w Fortranie algorytm makroprocesora STAGE2.

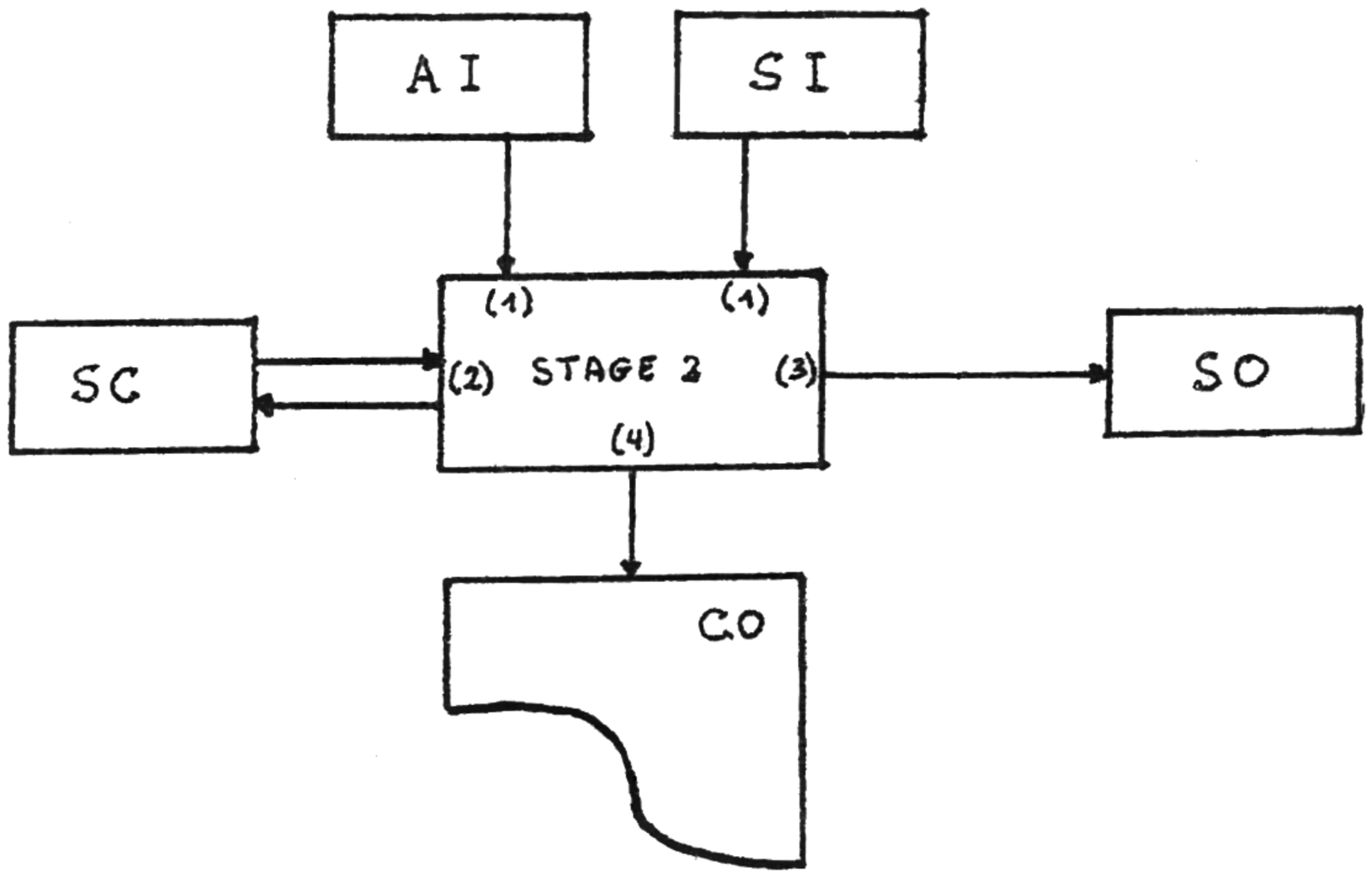
```
.'%'
FLG ' = '.
      JF'10 = JF'20%
%
''' ' = ' ' '.
      J'10'40 = J'10'50 '60 J'10'70%
%
''' ' = ''' '.
      J'10'40 = J'50'80%
%
GET ' = '.
      JF'10 = L (JP'20)%
      JV'10 = L (JP'20+1)%
      JP'10 = L (JP'20+2)%
%
STO ' = '.
      L(JP'10)=JF'20%
      L(JP'10+1)=JV'20%
      L(JP'10+2)=JP'20%
%
TO '' IF ''' ' = '.
      IF (J'30'60 .EQ. J'30'70) GOTO '10'20%
%
TO '' IF ''' ' '' '.
      IF (J'30'60 .'70'80. J'30'90) GOTO '10'20%
%
TO '' BY '.
      ASSIGN '00 TO JP'30%
      GOTO '10'20%
  '00 CONTINUE%
%
RETURN BY '.
      GOTO JP'10, (100)
%
TO ''.
      GOTO '10'20%
%
STOP.
      RETURN%
%
VAL ' = CHAR.
      JV'10 = LB (LBR)%
      LBR = LBR + 1%
%
CHAR = VAL '.
      CALL IWRCH (JV'10, LB, LBW, JF'10, LBL)%
%
```

```
READ NEXT '.
      JF'10 = IOOP (-1, JV'10, LB, 1, LBL)%
      LB (LBL) = -1%
      LBR = 1%
%
WRITE NEXT '.
      JF'10 = IOOP (1, JV'10, LB, 1, LBL)%
      LBW = 1%
%
REWIND '.
      JF'10 = IOOP (0, JV'10, LB, 1, 1)%
      JF'10 = 0%
%
LOC ''.
   '10'20 CONTINUE%
%
MESSAGE '''' TO '.
      MB (11) = '11%
      MB (12) = '21%
      MB (13) = '31%
      MB (14) = '41%
      JF'50 = IOOP (1, JV'50, MB, 1, 21)%
%
END PROGRAM.
      END%
%
MESSAGE ''''''.
      DO 1000 J = 1, 9%
 1000 MB (J) = '11%
      MB (10) = '21%
      MB (15) = '21%
      MB (16) = '31%
      MB (17) = '41%
      MB (18) = '51%
      MB (19) = '61%
      MB (20) = '71%
%%
```

```
C
      SUBROUTINE PROGR
C     ================
      DIMENSION LB (81), MB (20), L(12000)
C
C     IMPLEMENTATION CONSTANTS
C
      JP7 = 3
      JP8 = 1
      JP9 = 12000
C
C     FLUB CONSTANTS
C
      JF0 = 0
      JF1 = 1
      JF2 = 2
      JF3 = 3
      JV0 = 0
      JV1 = 1
      JV2 = 2
      JV3 = 3
      JV4 = 4
      JV5 = 5
      JV6 = 6
      JV7 = 7
      JV8 = 8
      JV9 = 9
      JP0 = 0
      JP1 = 1
      JP2 = 2
      JP3 = 3
      JP5 = 10
C
      LBL = 1
      LBR = 1
      LBW = 1
C
C     THE MAXIMAL FLUB ADDRESS VALUE
C
      JP9 = JP8 + (JP9 / JP7 - 1) * JP7
C
C     START OF FLUB
C
MESSAGE * ERROR.
```

A2. Opis dystrybuowanej wersji Makroprocesora STAGE2

Makroprocesor pracuje na minikomputerze MERA 400 pod systemem SOM3 w konfiguracji przedstawionej na rys. A2.1.



rys. A2.1
Konfiguracja makroprocesora
STAGE2

Opcja AO pozwala na czytanie początkowego fragmentu danych ze strumienia AI. Po przeczytaniu końca pliku następuje automatyczne przełączenie na dalsze czytanie ze strumienia SI.

Dystrybuowana jest:

1. binarna wersja S makroprocesora (patrz rozdz. 5)
2. fortranowska wersja procedur WE/WY
3. tekst makroprocesora STAGE2 w języku FLUB.

Literatura

1. Aho A.V., Ullman J.D.: Principles of Compiler Design. Addison - Wesley Publishing Company, Reading, Massachusetts 1977.

2. Brown P.J.: Macro Processors and Techniques for Portable Software. John Wiley & Sons Ltd., 1974.
Wydanie polskie: Makrogeneratory i Oprogramowanie Przenośne. WNT Warszawa 1978.

3. Feuerhahn H., Koster C.H.A.: Static Semantic Checks in an Open-Ended Language. Nijmegen University, report no 6, 1977.

4. Gennart P.E. i inni: Portability of an Algol 68 Compiler. In: Machine-oriented Higher Level Languages, North Holland Publishing Company 1974.

5. Gries D.: Compiler Construction for Digital Computers. John Wiley & Sons Inc., 1971.

6. Iglewski M., Kuta M., Matwin S.: Mobilny System Programowania. Prace COPAN, nr 171, 1974.

7. Jojczyk K., Król J.: Mobilny Preprocesor Fortranu Strukturalnego. Podstawy Sterowania 10, 1, 1980.

8. Jojczyk K.: Makroprocesor ogólnego przeznaczenia STAGE2. Raport IIUJ (w przygotowaniu).

9. Koster C.H.A.: CDL - A Compiler Implementation Language. In: Methods of Algorithmic Language Implementation, Springer Verlag 1977.

10. Nikliński P., Piechowicz J.: Konwersja programów symbolicznych Assembler 360 - Plan 4. Praca magisterska IIUJ 1979.

11. Orgas R.J. i inni: Wisp - A Self-Compiling List Processing Language. Basser Computing Dept., Technical report no. 36, Sydney 1965.

12. Papakonstantinou G.: A Portable Preprocessor for Recursive Fortran. Angewandte Informatik 11, 1975.

13. Papakonstantinou G.: A Poor Mans Realization of Attribute Grammars. Software Practice and Experience, 9, 9, 1979.

14. Płoski Z.: Makrogenerator STAGE2. Realizacja dla maszyny cyfrowej ODRA 1305. IIUWroc., raport nr P-7, grudzień 1977.

15. Stahl H.M.: Portability and Efficiency in Using an Open - Ended Language. Angewandte Informatik 5, 1979.

16. Waite W.M.: A Language Independent Macro Processor. Comm. ACM. 10, 7, 1970.

17. Waite W.M.: The Mobile Programming System: STAGE2. Comm. ACM, 13, 7, 1970.

18. Waite W.M.: Implementing Software for Non-Numeric Applications. Prentice Hall 1973.

Spis treści

1.	Wprowadzenie	3
2.	Opis typów modyfikacji parametrów	9
3.	Opis funkcji procesora	14
4.	Maszyna abstrakcyjna FLUB	16
5.	Implementacja makroprocesora STAGE2 na minikomputerze MERA 400	19
6.	STAGE2 jako język o zredukowanym jądrze	26
7.	Zastosowania	28
A1.	Składnia instrukcji maszyny abstrakcyjnej FLUB	31
A2.	Opis dystrybuowanej wersji makroprocesora STAGE2	36
	Literatura	37

S P R A W O Z D A N I A

I N S T Y T U T U I N F O R M A T Y K I U W

No 93 1980

- -

The portable software for minicomputer MERA-400
part one
The macroprocessor STAGE2

by Piotr Findeisen and Kazimierz Jojczyk

Summary

In this paper the implementation of mobile programming system STAGE2 for minicomputer MERA 400 is discussed. This work is a part of a complex project realized at the universities of Warsaw and Cracow. A few remarks about STAGE2 open-endness and the review of its application are also included.